Auxiliary Functions:

batchnorm_backward(dA, batchnorm_cache):

In order to backpropagate through the batch norm layer we need to calculate $\frac{\partial L}{\partial A}$.
We already have $\frac{\partial L}{\partial \hat{A}}$ and according to the chain rule we get: $\frac{\partial L}{\partial A} = \frac{\partial L}{\partial \hat{A}} * \frac{\partial \hat{A}}{\partial A}$.
Therefore we just need to calculate $\frac{\partial \hat{A}}{\partial A}$.

https://kratzert.github.io/2016/02/12/understanding-the-gradient-flow-through-the-batch-normalization-layer.html

split_into_batches(X, Y, batch_size):

For each epoch we need to iterate over the whole training data, on one mini-batch at a time. This function split the training data into mini-batches according to the batch_size parameter. The split is random over all the permutations of X and Y.

Code Decisions Explanation:

Initialization of W and b – I tried several methods of initialization for W such as Xavier, Kaiming, Xavier/2, Kaiming/2 and other simpler methods. I've read that Xavier is better for non-linear activation functions like sigmoid, softmax and tanh, while Kaiming is better for linear activation functions like relu, so I tried to combine them correctly but even this method wasn't good as the simple method. Therefore, the following results are based on the simple initialization that is in the submitted code.

Softmax – I've encountered many overflow errors due to the nature of the softmax function, so I changed it to be shifted by the max value of the input so the softmax is applied on numbers at most 0 (We can change it to be shifted less, so that the biggest number is 700 because e^{700} is still not overflowing).

Compute_cost – added eps in the log to solve numerical issues.

Linear_backward – we get as inputs: $A_{prev}, W, dZ = \frac{\partial L}{\partial Z}$

We know that $dW = \frac{\partial L}{\partial W} = \frac{\partial L}{\partial Z} * \frac{\partial Z}{\partial W} = dZ * A_{prev}$

We know that $dA_{prev} = \frac{\partial L}{\partial A_{prev}} = \frac{\partial L}{\partial Z} * \frac{\partial Z}{\partial A_{prev}} = dZ * W$

We know that $db = \frac{\partial L}{\partial b} = \frac{\partial L}{dZ} * \frac{\partial Z}{\partial b} = dZ * 1 = dZ$

Softmax_backward – AL is the output of the network. This function should return AL-Y, and the input it gets is dA which is basically AL. But I don't have access to Y there, so from the pervious function (L_model_backward) we compute AL-Y and send it to linear_activation_backward and then to softmax_backward such that dA=AL-Y.

Experiments:

The following results were tested across many batch sizes and weight decays.
Batch sizes = [32,64,128,250,500,1000,2000]
Weight decays = [0.1,0.01,0.001,0.0005,0.0001,0.00001,0.000001,0.0000005]
The next results will include only the best parameters.

** Note that the following results are with the stopping criterion of 0.001, but without it, the accuracy is 94%-95% after 50-100 epochs every time (the stopping criterion is weird, sometimes the accuracy is decreasing and it's ok, we should continue training for better results).

Experiment 1 – without normalization:

batch sizes: 1000 (also 250 and 500 were good, but without a major difference).

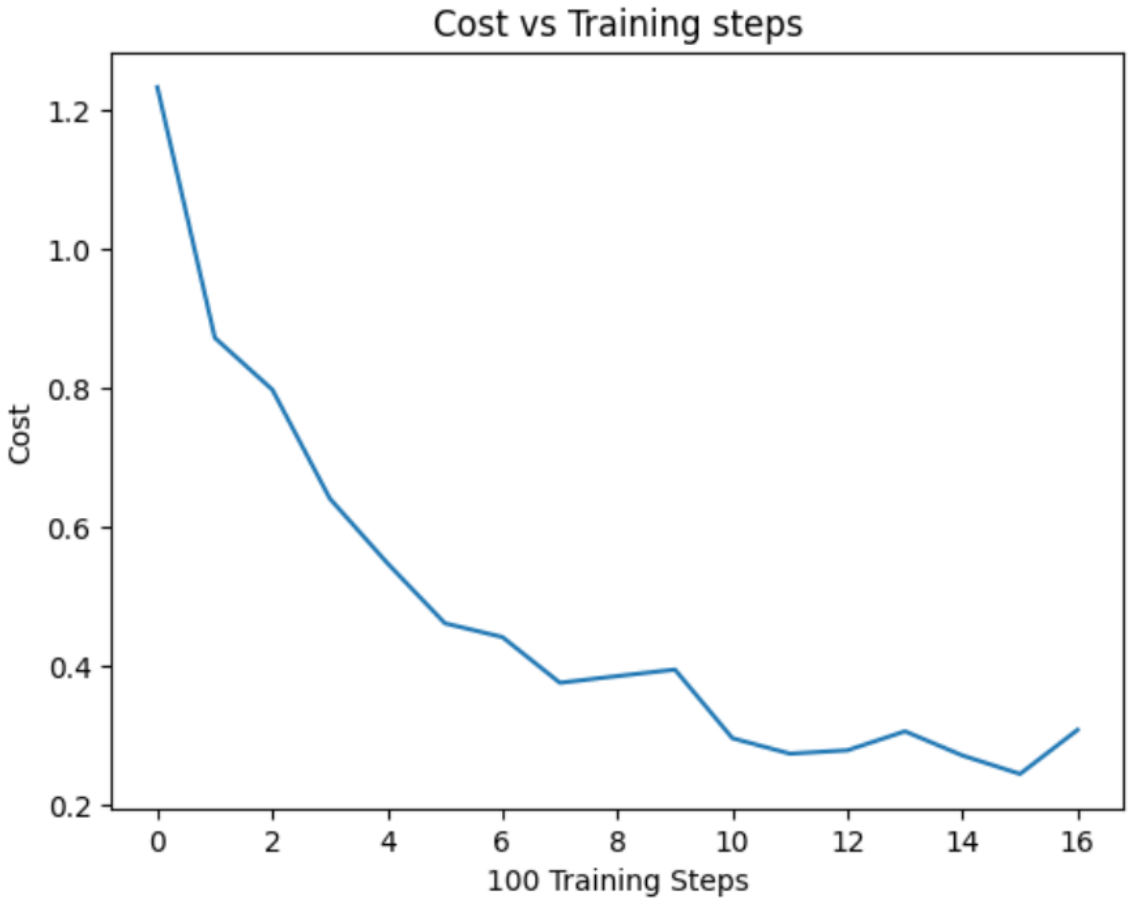
number of iterations: 48

number of epochs: 33 (the number of epochs until convergence was unstable, it may change according to the initialization randomization and the stopping criterion).

train final accuracy: 0.9087

test final accuracy: 0.9036

validation final accuracy: 0.8980



Experiment 2 – with batch normalization:

batch sizes: 1000 (also 250 and 500 were good, but without a major difference).

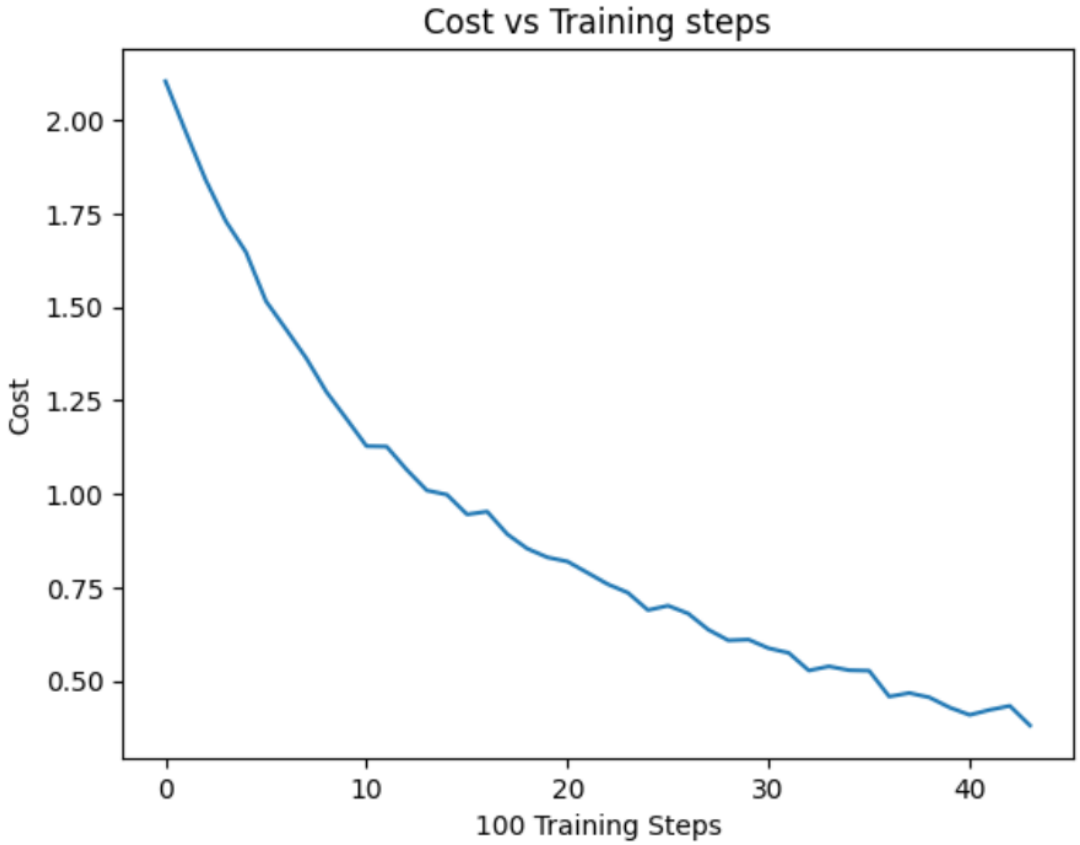
number of iterations: 48

number of epochs: 91 (the number of epochs until convergence was unstable, it may change according to the initialization randomization and the stopping criterion).

train final accuracy: 0.9242

test final accuracy: 0.9173

validation final accuracy: 0.9141



Changes explanation:

L_model_forward – after relu activations if using batchnorm, then apply batch normalization (X-mean(X))/std(X+epsilon). In addition, it's important to keep in cache the values before the normalization in order to compute their gradient in the backpropagation process. Therefore, the method should also return the batchnorm caches.

Linear_activation_backward – before relu gradient computation if using batchnorm, then compute the gradient of the batch normalization layer. Therefore, the method should also get the current batchnorm cache.

L_model_backward – if using batchnorm then pass the current batchnorm cache to the Linear_activation_backward function when getting the gradient for relu.

Experiment 3 – with L2 normalization:

Weight decay: 0.0005

batch sizes: 500 (also 250 and 1000 were good, but without a major difference).

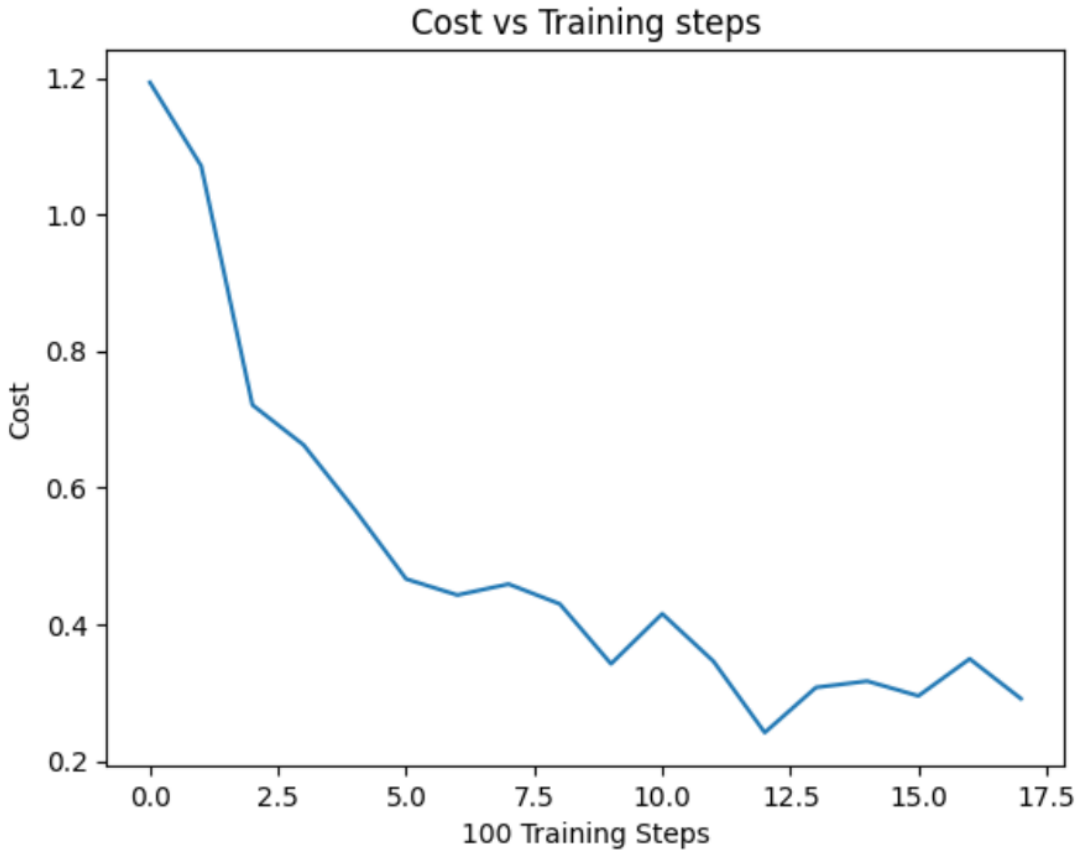
number of iterations: 96

number of epochs: 18 (the number of epochs until convergence was unstable, it may change according to the initialization randomization and the stopping criterion).

train final accuracy: 0.9240

test final accuracy: 0.9232

validation final accuracy: 0.9193



Changes explanation:

Compute_cost – if using L2 normalization then add omega to the current cost $\left(\Omega = \frac{weight\ decay}{2} * W^T W\right)$.

Linear_backward – if using L2 normalization then add the $weight\ decay * W$ to dW.

Update_parameters - if using L2 normalization then:

$W_{new} = (1 - learning\ rate * weight\ decay) * W - learning\ rate * grad(W)$.

Comparison:

	Test Final Accuracy	Final Cost	Running Time	#Training Steps
Experiment 1	0.9036	0.3086	22.78 (sec)	1584
Experiment 2	0.9173	0.3807	70.04 (sec)	4368
Experiment 3	0.9232	0.2912	16.23 (sec)	1728

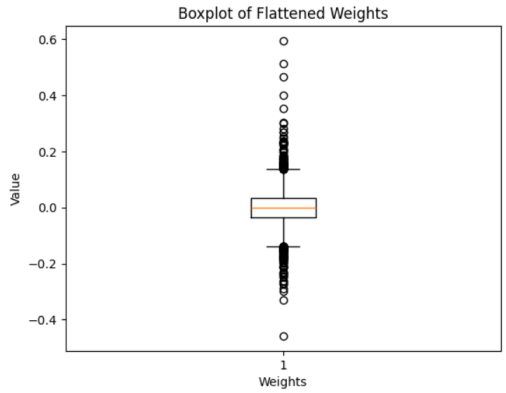
We can see that experiment 3 is better than 2 that is better than 1. Except to the number training step, but experiment 3 is based on batch size of 500, which mean that at every epoch the number of training steps is double the number of experiments 2 and 1. We can see that in the number of epochs needed to converge for experiment 3 is the lowest.

Although runs of experiments 1 were sometimes better than runs of experiments 2 and 3, and runs of experiments 2 were sometimes better than runs of experiments 3, the overall results (averages results) show that using L2 normalization is improving the learning process compared to batch normalization and without normalization. And using batch normalization is better than not using normalization at all.

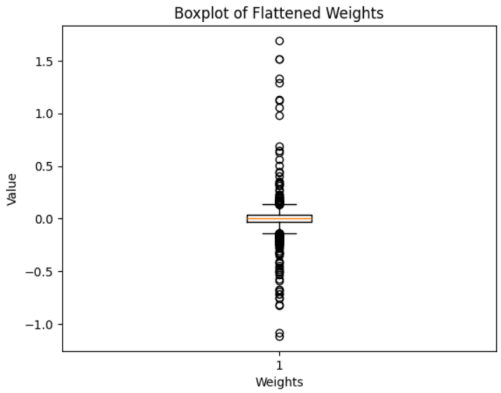
These results can be explained by the fact that L2 normalization is directly regularizes the model, leading to better generalization. Batch normalization improve the stability and gradient flow but didn't regularize the model as effective as L2 normalization, especially because we don't use beta and gamma. And furthermore, batch normalization is generally better if applying before the activation function and after the linear function. Without normalization at all, the model suffers from internal covariate shift and gradient issues, resulting in the poorest performance. I must mention that the differences between the experiments are minor and all of them performed well on certain runs.

Weights:

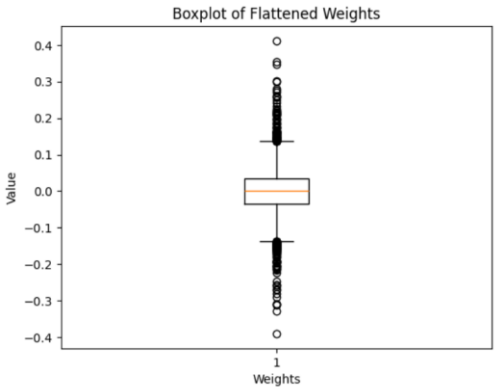
Without Normalization:



Batch Normalization:



L2 Normalization:



We can see that as for batch normalization and without it, the weights aren't normalized as for L2 normalization, in which the weights are spread more equally between -1 and 1.

In addition, we can see that the scaling is different, with L2 the biggest weight is 0.4, while without it there are weights of 0.6. And with batch normalization it can even be almost 2. Therefore, weights in L2 normalization are reaching closer to 0 but not exactly 0, which makes them more manageable when training.